5 CONCLUSIONS AND FUTURE WORK

You're bound to be unhappy if you optimize everything.

— Donald Knuth

THE growing adoption of WebAssembly requires efficent code analysis and hardening techniques. This thesis contributes to this effort with a comprehensive set of methods and tools for Software Diversification in WebAssembly. We introduce three technical contributions in this dissertation: CROW, MEWE, and WASM-MUTATE. Additionally, we present specific use cases for exploiting the diversification created for WebAssembly programs. In this chapter, we summarize the technical contributions of this dissertation, including an overview of the empirical findings of our research. Finally, we discuss future research directions in WebAssembly Software Diversification.

5.1 Summary of technical contributions

TODO Mention that CROW can only affect peephole and deterministic code.

This thesis expands the field of Software Diversification within WebAssembly by implementing two distinct methods: compiler-based and binary-based approaches. Taking source code and LLVM bitcode as input, the compiler-based method generates WebAssembly variants. It uses enumerative synthesis and SMT solvers to produce numerous functionally equivalent variants. Importantly, these generated variants can be converted into multivariant binaries, thus enabling execution path randomization. Our compiler-based approach specializes in producing high-preservation variants, indicating that considerable effort is necessary to revert these variants to their initial program. However, a bottleneck in the variant generation process, specifically the use of SMT solvers for functional verification, undermines it when compared with the binary-based method. Furthermore, this method can only modify the code and function sections of WebAssembly binaries.

[0] Compilation probe time 2023/11/17 18:22:13

On the other hand, the method based on binary utilizes random e-graph traversals to create variants. This approach eliminates the need for modifications to existing compilers, ensuring compatibility with all existing WebAssembly binaries. Additionally, it offers a swift, efficient and novel method for generating variants through inexpensive random e-graph traversals. Consequently, our binary-based approach can produce variants at a scale at least one order of magnitude larger than our compiler-based approach. The binary-based method can generate variants by transforming any segment of the Wasm binary. However, the preservation of the generated variants is lower than the compiler-based approach.

We have developed three open-source tools that are publicly accessible: CROW, MEWE, and WASM-MUTATE. CROW and MEWE utilize a compiler-based approach, whereas WASM-MUTATE employs a method based on binary. These tools automate the process of diversification, thereby increasing their practicality for deployment. At present, WASM-MUTATE is enhancing WebAssembly compilers[1] in real-world situations. Our tools are complementary, providing combined utility. For instance, when the source code for a WebAssembly binary is unavailable, WASM-MUTATE offers an efficient solution for the generation of code variants. On the other hand, CROW and MEWE are particularly suited for scenarios that require a high level of variant preservation. Finally, one can use CROW and MEWE to generate a set of variants, which can then serve as rewriting rules for WASM-MUTATE. Moreover, when practitioners need to quickly generate variants, they could employ WASM-MUTATE, despite a potential decrease in the preservation of variants.

5.2 Summary of empirical findings

We demonstrate the practical application of Offensive Software Diversification in WebAssembly. In particular, we diversify 33 WebAssembly cryptomalwares automatically, generating numerous variants in just a few minutes. These variants successfully evade detection by state-of-the-art malware detection systems. Our research confirms the existence of opportunities for the malware detection community to strengthen the automatic detection of cryptojacking WebAssembly malware. Specifically, developers can improve the detection of WebAssembly malware by using multiple malware oracles, or meta-oracles. Additionally, these practitioners could employ feedback-guided diversification to identify specific transformations their implementation is susceptible to. For instance, our study found that the addition of arbitrary custom sections to WebAssembly binaries is a highly effective transformation for evading detection. In practice, no WebAssembly engine uses custom sections, so injecting them does not impact the performance of the WebAssembly binary. Developers could enhance

[1] `https://github.com/bytecodealliance/wasm-tools`

their detection systems' robustness by normalizing the WebAssembly binaries, removing custom sections. This logic also applies to other transformations, such as adding unreachable code, another effective method for evading detection.

On the other side of the coin, our techniques enhance overall security from a Defensive Software Diversification perspective. We facilitate the deployment of unique, diversified and hardened WebAssembly binaries. As previously demonstrated, WebAssembly variants produced by our tools exhibit improved resistance to side-channel attacks. Our tools generate variants by modifying malicious code patterns such as embedded timers used to conduct timing side-channel attacks. Simultaneously, they can produce variants that introduce noise into the execution side-channels of the original program, concurrently altering the memory layout of the JITed code generated by the host engine.

Our methods remarkably generate thousands of variants in mere minutes. The swift production of these variants is due to the rapid transformation of WebAssembly binaries. This swift generation of variants is particularly advantageous in highly dynamic scenarios such as FaaS and CDN platforms. We have empirically tested the effectiveness of moving target defense techniques[**?**] on the Fastly edge computing platform. In this scenario, we incorporate multivariant executions[**?**]. Fastly can redeploy a WebAssembly binary across its 73 datacenters worldwide in 13 seconds on average. This enables the practical deployment of a unique variant per node using our tools. However, a 13-second window may still pose a risk despite each node potentially hosting a distinct WebAssembly variant. To mitigate this, we use multivariant binaries, invoking a unique variant with each execution. Our tools can generate dozens of unique variants every few seconds, each serving as a multivariant binary packaging thousands of other variants. This illustrates the real-world application of Defensive Software Diversification to a WebAssembly standalone scenario.

5.3 Future Work

Along with this dissertation we have highlighted several open challenges related to Software Diversification in WebAssembly. These challenges open up several directions for future research. In the following, we outline two concrete directions.

Improving WebAssembly malware detection: Malware detection is a well-known and challenging issue [**?**]. The problem intensifies when considering malware detection to be predictable, especially when the malware is assumed to be identically replicated across victims. We have successfully used Software Diversification to evade malware detection. This shows that, if applicable, predictability can be exploited by attackers using Software Diversification for offensive purposes. In previous discussions, we have detailed the benefits of feedback-guided diversification. This method aids in discerning specific

transformations for which defense systems are more or less effective. In future works, we plan to explore this approach further.

In addition, our current work elucidates the potential effectiveness of program normalization in enhancing the accuracy of malware detection systems. By comparing the canonical program with an existing dataset of malware, we could rapidly and efficiently detect malware. Specifically, we could employ a practically costless process of pre-compiling Wasm binaries as a preparatory measure for malware classifiers. For instance, a Wasm binary could first be JITed to machine code to effectively reduce the malware variant space. This method could significantly boost the efficiency and precision of malware detection systems. Yet, it heavily relies on the low preservation of the generated variants, i.e., highly preserved malware variants might be difficult to normalize, and thus to detect. We believe such an issue could be solved by data augmentation techniques.

Dataset augmentation has been demonstrated to enhance the precision of classification models [**?** **?** **?**]. Therefore, our tools could be utilized to produce a wide array of known malware variants. Such variants could serve to augment the dataset employed in the training of malware detection systems. This could potentially rectify the inherent issue of the high preservation of malware variants. Specifically, although the canonical program may not be located within a reasonable time for a highly preserved malware variant, a closely related variant may have already been incorporated into the augmented dataset. In summary, this method could bolster the robustness of the detection systems, thereby making them more resistant to evasion techniques.

Feedback-guided Diversification: As presented in Chapter 4, feedback-guided diversification can facilitate the identification of specific transformations aiding particular objectives such as malware evasion and side-channel protection. On the contrary, stochastic diversification may generate variants that do not align with the specific objective. For instance, our approaches have shown less impact on ret2spec and pht side-channel attacks compared to btb attacks when using stochastic diversification.

This issue can e seen as a search space exploration problem. The space can be divided to better focus the diversification process. The primary advantage of this division is the reduction of the search space, consequently accelerating and refining the diversification process. We aim to delve further into this approach, potentially using feedback-guided based on specific code patterns to disrupt, e.g., embedded timers. This strategy could be applied to other situations, such as mitigating specific side-channels like port contention [**?**].

We anticipate several challenges in the context of WebAssembly. Practically, one should balance critical dimensions: the number of variants, the generation speed of the variants, the preservation of the variants, the diversification objectives, and the performance of the generated variants. For instance, high preservation might inadvertently allow vulnerable code to persist. Even though we can generate more resilient variants, the compiler's role in mitigating initial

vulnerabilities is essential. In concrete, we could generate variants with disrupted malicious code patterns, but the compiler could still generate vulnerable code by reversing it. High preservation could therefore potentially impede the removal of potential security weaknesses during the compilation process.

In future research, we aim to examine the application of feedback-guided diversification by incorporating these dimensions into the feedback loop. For instance, we may use a variety of WebAssembly JIT compilers[**?**] to assess the preservation of the generated variants. Furthermore, the influence of specific code patterns may guide the diversification process. In the context of WebAssembly, we might utilize the broad and growing range of analysis tools to strengthen feedback. For instance, we could employ MINOS [**?**] for rapid binary classification and VeriWasm[**?**] for counting security smell patterns. In other words, we envision meta-oracles capable of providing feedback on the strength of the diversification process. The integration of such feedback into the creation of the WebAssembly variants awaits exploration.